Prototyping a distributed objet-oriented operating system on UNIX

Marc Shapiro

INRIA

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1082

Programme 3
Réseaux et Systèmes Répartis

PROTOTYPING A DISTRIBUTED OBJET-ORIENTED OPERATING SYSTEM ON UNIX

Marc SHAPIRO

Août 1989

RR-1082

Prototyping a distributed object-oriented operating system on Unix

Prototypage d'un système d'exploitation réparti à objets sur Unix

Marc Shapiro

Août 1989

This is a reprint of an article presented at the Workshop on Experiences with Building Distributed and Multiprocessor Systems (WEBDMS), Ft. Lauderdale FL (USA), October 1989.

Nous reproduisons ici un article présenté au "Workshop on Experiences with Building Distributed and Multiprocessor Systems" (WEBDMS), Ft. Lauderdale FL (USA), octobre 1989.

Abstract

The SOR group at INRIA has built a prototype distributed object-oriented operating system, called SOS, on top of Unix. SOS is based on migratable medium-grained elementary-objects, on top of which all its other basic mechanisms (such as composite objects, dynamic linking, and dynamic type-checking) are built.

SOS supports distributed or "fragmented" objects. A fragmented object is created by spreading out *proxies* from a *provider*. The public interface of the fragmented object is provided locally by proxies. Proxies may communicate directly, without going through the public interface (via messages, sharing, or any other means).

The most positive accomplishment of SOS is its elementary-object concept. A programmer-defined object can impose its own semantics or policies on system-implemented mechanisms, thanks to our *upcall* and *prerequisite* mechanisms. For instance object migration and storage are performed under the control of the system, but they respect the semantics of each individual object.

One negative aspect is that a fragmented object can only be created dynamically. Other problems arise from the protoyping environment of Unix and C++.

Résumé

Le groupe SOR de l'INRIA a construit, sur Unix, un prototype de système d'exploitation à objets, appelé SOS. SOS est basé sur des objets élémentaires migrants, de granularité moyenne. Sur cette base, sont construits tous ses autres mécanismes, tels les objets composites, l'édition de liens dynamique, et la vérification de type dynamique.

SOS réalise une notion d'objets répartis (ou "fragmentés"). Un objet réparti est créé en *essaimant* des *mandataires* à partir d'un *fournisseur*. L'interface publique de l'objet réparti est offerte localement par les mandataires. Les mandataires peuvent communiquer directement entre eux, sans passer par l'interface publique, par messages, par partage, ou tout autre moyen.

Le résultat le plus positif de SOS est sa notion d'objet élémentaire. Un objet peut imposer sa sémantique aux opérations système qui le concernent, grâce aux mécanismes d'*appel montant* et de prérequis. Ainsi, la migration et le stockage sont réalisés par le système, mais en respectant la sémantique des objets.

Un aspect négatif est que les objets répartis ne peuvent être créés que dynamiquement. D'autres problèmes proviennent de l'environnement de prototypage, Unix et C++.

Contents

1 Introduction

The object-oriented programming methodology is becoming increasingly popular, for all sorts of applications. Many object-oriented programming languages exist, such as Smalltalk [9], C++ [24], Eiffel [15], CLOS [7], etc. Each compiler enforces its own object model, and deals with the inadequacies of existing operating systems in its own way.

The goal of the SOR group of INRIA is to implement an object management support layer common to all applications and languages. This should: facilitate the implementation of object-oriented language compilers; make applications more efficient; allow independent applications to communicate and share objects, without prior arrangement.

The services of the common object management support layer include support for creating, deleting, migrating, storing, localizing, and invoking objects. If these services are sufficiently complete, low-level, generic, language-independent, application-independent, and efficient, then they can legitimately be called an *object-oriented operating system.*

Within the office-workstation Esprit project SOMIW (1985–1988) we have built a prototype called SOS. It has been used for the SOMIW applications, such as BFIR2, a multimedia document toolbox, and Images, an user-interface management system. SOS is written in C++ and prototyped on top of Unix.

SOS supports an elementary-object model which is both simple and powerful. A reasonable granularity is of the order of a hundred bytes and up per object. Composite objects, object storage, dynamic linking and dynamic type-checking are built on top of elementary-object mechanisms.

In addition, SOS extends the object concept to distributed, or "fragmented", objects. The public interface of the fragmented object is provided locally by its fragments, which are the elementary objects. This encourages the structured design of distributed applications based on the "Proxy Principle" [19]. All the SOS system services (for instance, the Name Service [11]) are built as fragmented objects with local proxy interfaces.

We now have accumulated enough experience to assess the SOS design and implementation. A most positive aspect is its elementary-object model. Fragmented objects have proved a good way to structure distributed

applications, although they are hard to use. A weakness of our implementation is that fragmented objects cannot be static or persistent. The proxy concept poses a protection problem. The implementation of SOS is not really language-independent. Object migration imposes restrictions on the use of C++ as the application programming language. The prototype is slow.

In the remainder of this paper we will explain some of the aspects of the SOS prototype. We expose design rationales and implementation, and discuss features and limitations. We start with a short comparison with similar work, in section 2. Section 3 first gives some background on SOS concepts and implementation. Section 4 is about elementary objects. It is followed by section 5, an explanation of fragmented objects. Follows section 6, which discusses object migration. Finally, in section 7, we give an assessment of the design and implementation of the prototype.

2 Comparison with similar work

Emerald is an object-oriented language for distributed programming, featuring fine-grained mobillity [10]. The compiler transforms the user-defined object representation in order to facilitate migration: its first few bytes are a standard descriptor, and all fields of a similar type are grouped together. Conceptually, all objects live in a single, network-wide address space. An object reference is global, but a local reference is optimized into a pointer.

In contrast, the SOS approach is operating-system based. We do not assume any standard representation. Instead, system information is well separated from programmer-defined data, and the system performs upcalls on objects. Instead of a single address space, we stress *structuring* the universe.

Choices [3] is a family of operating systems built using object-oriented design. The services it exports to applications are fairly conventional. The emphasis in SOS was not its internal design, but providing new services to facilitate the implementation of distributed object-based applications.

Clouds [5] is another object-oriented OS. Its emphasis is on integrating support for reliable objects in the low level of the system. (Our current

design has no particular provisions for reliability.) Their objects are presumably much larger-grained than ours, since a Clouds object executes in its own address space.

Guide/Comandos [6] is a language-driven distributed programming environment. The universe is structured in separate, multi-machine address spaces called domains. When a domain needs access to an object located on a remote machine, it extends itself to that machine, and maps the object in. This structure is easier to use than SOS's fragmented objects; however the latter scales better, and deals better with replication.

Gothic [1], a language and system for reliable distributed programs, is based on a theory of fragmented objects invoked via "multi-functions" (side-effect free invocations, with co-ordinated multiple threads), supported by the language. Our fragmented objects are ad-hoc but more flexible.

3 Background

3.1 SOS concepts

SOS is an object-oriented operating system. It provides support for arbitrary user-defined objects, including object creation, destruction, migration, storage, localization, communication, naming, etc.

An *elementary object* is an user-defined data segment with a system descriptor.[1] Considering the descriptor overhead, a reasonable granularity for the data segment is a size of 50 or 100 bytes and up.

We assume that the data was created using an object-oriented programming language compiler, and that it is accessed only via its type-checked procedural interface. An object accesses system services by calling the appropriate primitives; we call this a *downcall*. Conversely, the system can invoke, with an *upcall*, a few well-known procedures of an object. For instance, a cross-context invocation (see below) is executed by upcalling the

[1] Composite objects, with multiple data segments connected by pointers, are built on top of elementary objects, but they will not be considered here. Similarly, object storage is built on top of migration. See [21].

stub procedure of the target object; each object has its own stub which can be redefined at will.

An object is designated by its address (within the context), or globally by a *reference* containing an OID (object identifier) and a location hint.

SOS comprises a kernel and system services running on top of it. The kernel provides separate address spaces (*contexts*), light-weight threads in a context (*tasks*), and inter-context communication. A context may contain any number of elementary objects. Elementary objects may migrate between contexts; at any point in time, an elementary object is active within a single context, or stored on disk.

SOS extends the object concept to distributed or *fragmented* objects (see figure 1). A fragmented object is implemented as a *group* of elementary objects located in different contexts; i.e. its representation is the reunion of the local "fragments".

Just as an elementary object can access its own representation, bypassing the procedural interface, similarly the individual fragments are allowed to use untyped communication to each other: cross-context invocation, communication protocols, shared memory, shared files, etc. Objects which are not fragments of a same group are not permitted to communicate in this manner.

A fragment may create and add a new fragment to the group, and export it to another context. Group membership is preserved across migration; thus the group grows by spreading.

Applications on SOS are designed according to the "Proxy Principle" [19]: to use some service, a client invokes a local *proxy* for the service, i.e. a local object which is a fragment of the group implementing that service. If such a proxy is not locally available, it must first be acquired by sending an import request to a *provider* for that group, i.e. a particular object in charge of delivering proxies.

For instance, for a graphics program to output to the screen, it will request a window proxy; the window manager is the provider for this resource. The window manager will reserve the window and allocate it to a window server object; it will create a window proxy which is exported to the graphics program. The proxy has a graphical interface (e.g. drawVector,

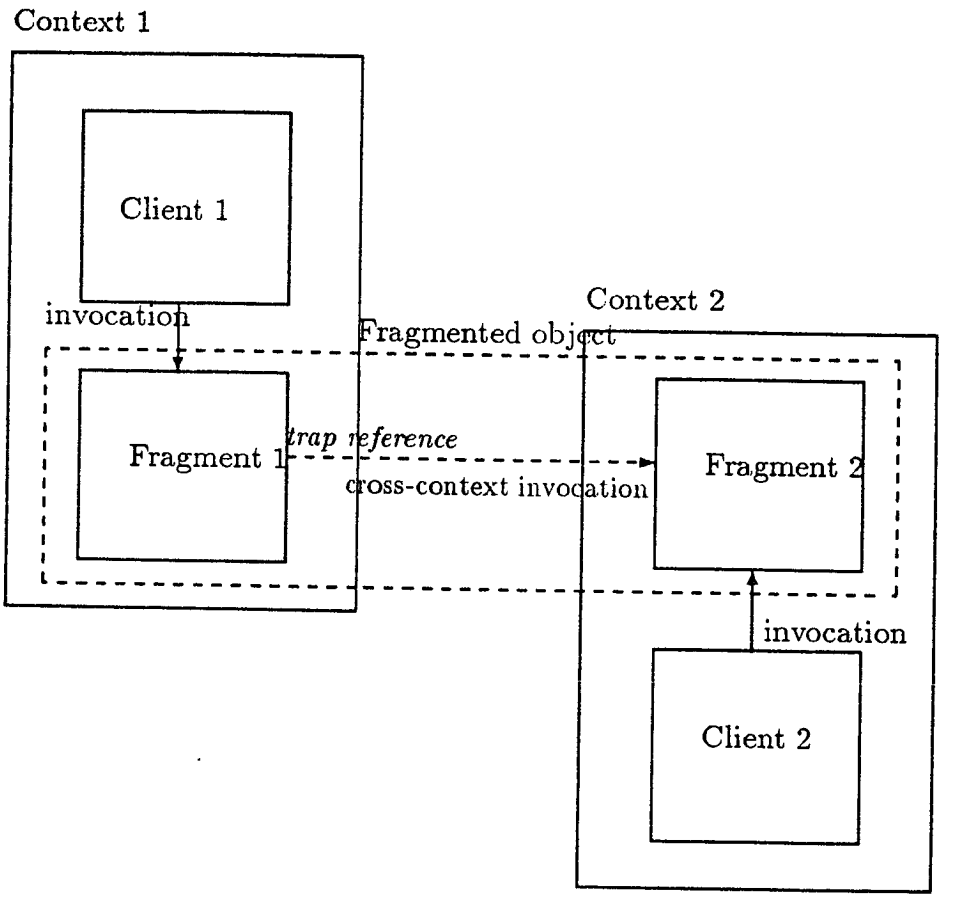

Figure 1: A fragmented object (group) with two fragments, used by two clients in different contexts.

displayCharacter, etc.). It may interpret some requests locally (e.g. getWindowSize); others will be buffered before being sent down a *channel* to the window server when appropriate.

In this article, we are only interested in the objects managed by SOS, which we will call *SOS objects*. Only objects which are intended to be migrated, stored, or remotely accessed need to be SOS objects; application programs are free to manage their other, internal, "plain" objects. In the remainder of this paper we use the word "object" for "SOS object".

3.2 The prototype

Our prototype is implemented in C++ on top of Unix (SunOS 3.4). This article describes SOS Prototype Version 4, which was delivered to the SOMIW partners in the Fall of 1988 [22,23].

SOS objects are instances of the predefined class sosObject (or of a compatible class). C++ has so-called virtual procedures, invoked indirectly via a table of procedures [8], itself accessed by a pointer in the object's data. A class may override the pre-defined actions of a procedure, by replacing the corresponding entry in the procedure table. Upcalls are performed via this table; unfortunately, this is not language-independent.

Separate address spaces are provided by Unix. Tasks are implemented as a library (the task library of C++ [25] with some additions). Context management is performed by a Unix process called sos. Inter-context communication uses Unix-domain stream sockets.

On each machine, sos automatically starts a number of system-service contexts, the local servers for: the Acquaintance Service (AS) or object manager, the Storage Service, the Name Service, and the Communication Service (CS). Each new context starts with a pre-installed proxy for the AS, which allows the application to import proxies of the other (system- or user-defined) services. Applications are run from the Unix shell or the debugger.

In the remainder of this paper, we will take a look at the design and implementation of the SOS prototype, and evaluate it in the light of our experience. We will concentrate on the aspects of distributed management

and communication of objects; for a more in-depth introduction to SOS, see [20,22]. With one exception (export, section 6.2.1), all the interfaces given here are implemented with the explained semantics. They have been in use in the prototype for at least one year.

4 Elementary objects

The basic entity managed by the SOS Acquaintance Service (i.e. the object manager) is the *elementary object*. We have made the elementary object as simple as possible, a "least common denominator" for all uses.

At any point in time, an elementary object exists in a single context on a single machine. Each elementary object is different from all others; it is characterized by its own unique identifier called its *concrete OID*. An elementary object is known to SOS by its descriptor, called Acquaintance Descriptor (AD). There is a table of AD's per context, managed by the context's AS proxy.

An AD for some object contains the following information (the items in italics have to do with migration and groups, and will be defined later):

- Its concrete OID and (possibly) a list of *group OID*'s,
- The reference of its code object and (possibly) a list of *prerequisites*,
- The address and size of its data segment,
- (Possibly) Its list of *trap references*.

The class code is a predefined class of elementary objects. A code instance holds the compiled code for some class. For instance the code for some user-defined class X is managed by the code instance code_for_X. The reference from the AD to the object's code is necessary for migration.

The following table[2]gives the downcall interface for elementary-object

[2]The pseudocode

a . b (c) → d

means: invoke procedure b of object a, with in argument c, and returning value d. If no object a is mentioned, then the procedure is a primitive (actually, a procedure of the kernel or of the object manager).

management. There are no upcalls.

Downcalls for elementary-object management	
(sosObject *constructor)*	Object creation
(sosObject *destructor)*	Object destruction
obj . setCodeRef (ref)	Set code reference of object
find (ref1, radius) → ref2	Search for object location
getAddress (ref) → obj	Translate global reference to local address
getReference (obj, OID) → ref	Translate address to global reference

4.1 Creation and destruction of elementary objects

In C++, creating an object triggers a chain of calls to *constructor* procedures, starting from from the actual implementation class, up to the root of the inheritence tree (in this case, sosObject), and back down to the implementation class. A constructor is a mix of compiler-generated and user-defined code. Memory for the object is allocated (by malloc) in the compiler-generated part of the implementation class constructor, and passed up as a parameter to the sosObject constructor.

Thus, there is no explicit primitive for object creation: it is subsumed by the the sosObject constructor. It allocates an unused AD, and fills it with a newly-allocated OID, and with the address and size of the data.

The size of the data is not explicitly available to the sosObject constructor: it is taken from the malloc header. The reference to the code object is not available either; the constructor sets it to nil. The other parts of the AD are also initialized to nil.

The implicit AS interface for object deletion is the *destructor* procedure of sosObject, called automatically when an instance is deleted. The destruction of a context or a processor crash deletes all the contained instances. We are currently designing a mechanism to propagate object-destruction events to dependents of an object.

4.2 Assessment

The rationale of the above design is that the object creation and destruction primitives are rendered transparent by the C++ inheritence, thus simpli-

fying the task of the application programmer. One drawback is that the system interface is not clearly identified and not language-independent.

Another problem is that the sosObject constructor does not have all the necessary information; for instance the size of the data is obtained by the malloc-header hack. Similarly, the code reference can not be set by the constructor; a separate call to setCodeRef is necessary prior to migration.

4.3 Other primitives for elementary-object management

The find procedure of the AS, given a reference to an object, finds the actual location of the object (possibly by asking all the AS proxies within the specified radius), and returns a reference containing that exact location. If the argument is a reference to a group (see below), the returned value is the reference of its closest fragment.

getReference and getAddress translate between local object addresses and global references. If getAddress is passed a reference of a group, it returns the address of its local proxy, if any. The OID argument of getReference allows to pick between a reference to the elementary object itself, or to its group.

5 Fragmented objects (groups)

A fragmented object is implementented as a *group* of elementary objects, called its fragments. (Or, alternatively, a group is a single object with a fragmented representation.) Clients of the group may access it locally by its strongly-typed procedural interface, provided by the proxy fragments; its public interface is a sort of "union" of its fragments'.

Each fragment of the group may access the object's internal representation; fragments may communicate via untyped shared memory or messages, for instance.

The group is conceptually a protection domain, entered by invoking a local proxy.

This is illustrated in figure 1.

The following table shows the interfaces for group management.

Group downcall interface	
addGroupOID (obj, OID) → index	Create a new group
obj1 . giveMyOID (obj2, index1)	Put obj2 in same group as obj1
obj1 . setTrapRef (obj2, opaque) → index1	Establish channel from obj1 to obj2
obj1 . giveTrapRef (obj2, index1, opaque) → index2	Duplicate channel
crossInvoke (callMsg, segs, index) → replyMsg	Send invocation, receive reply on channel
Group upcall interface	
stub (callMsg, segs) → replyMsg	Receive invocation, return reply

A group is characterized by the fact that each fragment carries the OID of the group, in addition to its concrete OID. An elementary object can be a fragment of zero, one, or more groups.

Members of a group enjoy mutual communication privileges, which are denied to non-fragments. An invocation *channel* is a unidirectional connection between two elementary objects on the same machine, materialized by a *trap reference* in the source object's AD pointing to the target.

Other types of communication within the group, such as shared files, are also available, but will not be detailed here. Shared memory should be possible, but we never implemented the appropriate interfaces.

5.1 Group management

The primitive addGroupOID assign a fresh group OID to an object, in order to start a new group.

A group is created implicitly by giveMyOID, which gives away an existing concrete or group OID (designated by its index in the list of OID's of obj1), to some object.

A group is destroyed when its last fragment goes away.

Channels are created by the primitives setTrapRef and giveTrapRef. The former procedure creates a channel between the current object and the first

argument; it returns the index of the channel in the current object's trap reference list.

giveTrapRef duplicates an existing channel: before the call, obj1 has a channel at index index1 to some receiver; after the call, obj2 also has a channel to the same receiver, at index index2.

As its name implies, the opaque argument is not interpreted by the system. It is simply stored at the sender end of the channel, and will be automatically prepended to every invocation sent on it. The receiver may test the opaque field of remote invocations to distinguish between its callers, and test their access rights.[3]

5.2 Cross-context invocation

The primitive crossInvoke sends an invocation on a channel of an elementary object, and returns a reply.

The arguments to crossInvoke are (in addition to the channel index) a message, and possibly a list of segment access rights. The message is of limited size (1024 bytes); any larger data is to be passed as a segment right.[4] Available rights are read, write, and create.

A cross-invocation causes an upcall to the stub procedure of the receiver within a newly-allocated task. The receiver gets a copy of the invocation message, and may access the segments according to the rights passed. stub returns a return message which is copied back to the caller. This procedure plays the role of Nelson's "client stub" [2].

5.3 Assessment

5.3.1 Constructing a group

Currently each fragment type is programmed "by hand", and there is no guarantee of consistency even within a particular type of group. We are

[3]The opaque attribute of a channel is similar in concept to the rights field of a capability in Amoeba [16] or Chorus [17].

[4]This is modeled after the V-System RPC [4].

working on a new tool, a "fragment generator" (similar to an RPC stub generator). It will take care of the common aspects of programming fragments and providers (viz. allocating group OID's, setting up channels, allocating message buffers, etc.). It will also allow to define group types with a well-defined structure, and enforce their internal consistency at compile time. Finally, it will provide help in coordinating state changes between fragments.

5.3.2 Protection

Only the currently-executing elementary object should have access to its own channels. The kernel attempts to enforce this, taking advantage of the fact that the C++ compiler adds a hidden argument to all invocations, which is the address of the invoked object. When crossInvoke is executed, the kernel gets the first argument of the penultimate stack frame, and considers it to be the current object (if it is a valid object address). The index argument is relative to that object.

Getting the current elementary object from the stack is a weak way of enforcing the group protection domain at run-time. Weak enforcement is acceptable, because groups are intended as a program structuring concept, not a confidentiality mechanism.

Given our environment (Unix and standard hardware), and the granularity of objects, it was unfeasible to implement a stronger form of run-time protection at a reasonable cost. A structured memory organization, like that offered by capability machines, might improve run-time protection. A more attractive idea is to enforce the integrity of the group at compile time. Our proposed "fragment generator" is a step in this direction.

To provide some protection of the group against spurious membership, giveMyOID and setTrapRef can only connect objects within the same context. The normal way of creating a group is to create proxies locally and migrate them (see below) to another context; group membership and channels are preserved across migration.

5.3.3 Static groups

There is also a need for static groups. For instance, a system service such as the AS, the Name Service, the Storage Service, or the Communication Service, is implemented by one server on each machine, which comes up at boot time. In order to communicate with its remote peers, it must already be a fragment of their group as soon it starts up. Currently, a protection loophole is needed to circumvent this problem: when the server comes up, it forges an OID with a given value (taken from a configuration file) and inserts itself in the group using addGroupOID.

This loophole should be protected by some privilege but in fact it is not. Better still, both the group and its fragments (the servers) should be *persistent*. SOS supports persistent objects, as a service above the basic mechanisms described here; a much tighter integration is needed to support persistent groups.

5.4 Communication protocols

The only communication protocol implemented by the kernel is a cross-context invocation along a trap-reference channel, within the same machine.

Remote (across machines) access and other protocols are performed by *protocol objects*, implemented by the Communication Service [12,13,14]. The CS offers a library of protocol types, such as multicast and stream protocols.

A protocol object is layered underneath the application object which it serves; this is illustrated by figure 2. A protocol object is itself a group of cooperating elementary protocol objects, instantiated in the communication service contexts of the individual machines. A channel is made to use a protocol by setting the trap reference to point to the appropriate elementary protocol object.

An elementary protocol object has two privileges: it can access the network, in order to implement remote communication; and it can be the source or the target of a trap reference, even though it is not a fragment of the application object. In all other respects, a communication object is a standard group.

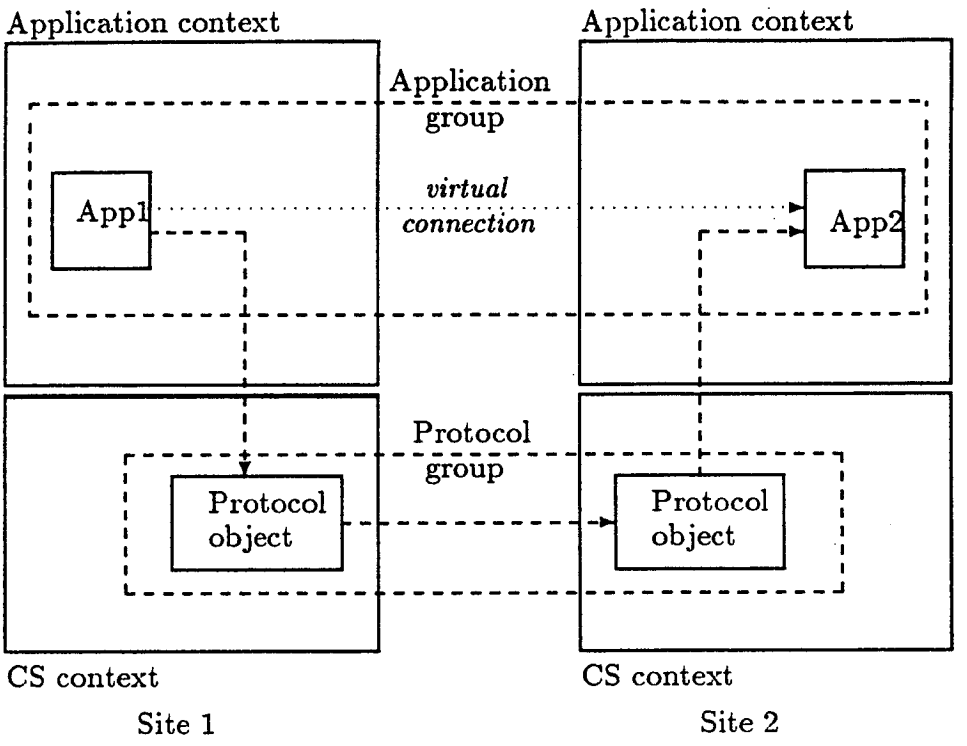

Figure 2: Fragmented protocol object, layered underneath the fragmented application object which it serves. The protocol object has the privilege of setting trap references which cross a group boundary.

6 Migration of elementary objects

A client gets access to a new service by getting a proxy for the service: the corresponding group migrates a fragment into the client's context. Migration is completely generic, thanks to appropriate upcalls (givePproxy starts an importation; the *reinitializer* finalizes a migration), and to the code and prerequisite objects.

Before explaining the migration interface, let us look at the migration algorithm.

6.1 Migration algorithm

Suppose elementary object X is to be migrated from source context S to destination context D. The algorithm starts when the decision to migrate X has been notified, and all access rights have been checked; we will ignore error cases. The algorithm is the following.

1. Make X unavailable to users in S.
2. Copy the AD of X from source to destination context. All the contents of the AD are preserved: its concrete OID, group OID's, trap references, code reference, prerequisite references, and size of data segment. However the address-of-data-segment field is invalid.
3. Using information in the AD, copy X's data from S to some arbitrary free location in D. Update the data address field in the destination AD.
4. If (a proxy of) X's code object is not yet present in D, import one. If present, skip this step. Similarly, import all prerequisites of X, if not already present.
5. Upcall the *re-initialization* procedure of X in D.
6. Make X available to users in D. The data and AD of X are destroyed in S.

7. The trap references of X have become invalid in D. The first time an invalidated trap reference is used for cross-invocation, the kernel executes special code to revalidate it, possibly in cooperation with the Communication Service.

The above describes the "move" variant of migration. The "copy" variant differs slightly: a new concrete OID is allocated in D (step 2), and the source copy is not destroyed, but instead is made available again (step 6).

6.2 Migration interface

The migration interface is given in the following table.

Migration downcall interface	
sosImport (key, importReq, "class", provider) → obj, procTable	Request import of obj of type class
new dynamic (provider) class (importReq, ...) → obj	Same, from C++ programs
obj2 . export (desc, index2)	Export described object along channel of obj2
obj . giveSelf () → desc	Use "move" semantics for migration of obj
obj . giveCopy () → desc	Use "copy" semantics for migration of obj
Migration upcall interface	
obj1 . giveProxy (importReq) → desc	import request for object described by desc
(re-initialization)	Finalize migration

There are two possibilities for migration: import and export. We will start with export, which is the simpler of the two.

6.2.1 Exporting

The call obj2 . export (desc, index) migrates an object obj1, described by desc, along the channel of obj2 indicated by index. Either obj1 . giveSelf () or obj1 . giveCopy () is used to prepare desc. Export uses the migration algorithm of section 6.1. The object on the other end of the channel will receive a special invocation message, signalling the arrival of an exported object.

6.2.2 Importing

The internal interface for requesting an import is sosImport. For C++ programmers, an easier-to-use interface is implemented by a compiler extension: the clause new dynamic (provider) class (importReq, ...) generates a call to sosImport followed by a call to the re-initializer. The arguments are: provider, the reference of an object which will be requested to provide a proxy, and importReq, an import request message carrying untyped request parameters. The AS adds to the import request the reference of the requestor. Possible extra arguments (indicated by the ellipsis) will be passed to the *re-initialization constructor.*

The other arguments to sosImport are automatically generated by the compiler: key describes the expected type of the imported object; and "class" is the name of the class in the new dynamic declaration, which is used to select a default provider.

The mechanics of importation are the following. The AS proxy of the requestor performs a find based on the provider reference. This yields the location of the provider object (or of one of its fragments if the reference was to a fragmented object). The AS proxy at that location then performs the giveProxy upcall on the provider, with a copy of the import request, carrying sufficient information to identify the requestor.

The provider's giveProxy selects some object M to be migrated, and calls either giveSelf or giveCopy, to prepare a description which it returns; alternatively, it may return an error indication. The object M could be the provider itself, or some other object of its context, or a stored object. In the latter case it must be of the same group. When giveProxy returns, M is migrated to the requestor, according to the algorithm of section 6.1.

At the end of a migration (step 5) a re-initialization procedure is upcalled, to allow finalization. A typical use of the re-initializer is to set pointers to meaningful values, or to request more importations.

6.3 Migration of code and prerequisites

We mentioned (step 4 of the migration algorithm of section 6.1) that the code and prerequisites are recursively imported, if not already present, be-

fore calling the re-initializer. The pre-requisites are the environment the migrated object needs in order to function; the object's code is just one kind of prerequisite. These are imported if not already present, in order to avoid waste: this allows two imported objects to share code if they are implemented similarly. The same mechanism supports static linking of the code for proxies, without any loss of functionality.

The giveProxy procedure for class code migrates a copy of the code. Two similarly-implemented objects in the same context share a single code object (because pre-requisites are imported only if not present).

Since a prerequisite is imported according to the same algorithm as other objects, its re-initialization procedure is called in step 5. The re-initializer for a code object is a *dynamic linker and type-checker.* The type is checked against the key argument to sosImport. The linking and type-checking algorithm are language-specific; other languages could be supported simply by implementing a new code class.

The strength of this design is that pre-requisites are elementary objects like any other. Dynamic linking and type-checking are automatic, without being wired in. The drawback is that type-checking is automatic only for the first import of an object of a certain class; type-checking for subsequent imports must be special-cased.

6.4 Assessment

We stress that the upcalls to giveProxy and to the initializer, together with pre-requisistes, implement a very important concept: extending a system-defined mechanism with *programmer-defined semantics.* Arbitrary objects can be migrated, and the semantics of their migration is type-specific, above a single, generic, system-implemented mechanism.

6.5 Calling the re-initializer

The C++ syntax for importation is an extension of the instantiation syntax, and in C++ the re-initializer is in fact a constructor. This is appropriate

since their semantics are quite similar.[5]

This raises the issue of whether the reinitializer call should be generated by the compiler, or performed by the system. The former solution permits extra arguments (in addition to the import request); the latter allows the system to know that re-initialization has succeeded. We opted for the compiler solution whenever possible, favoring the confort of C++ programmers. However for exports and for pre-requisite imports, the reinitializer can only be called by the system, hence the dual interface.

The compiler decision was bad for two reasons. First, it imposes to treat exports and pre-requisites differently from imports, which is confusing for the users. Second, and more importantly, an operating system should be independent of a particular language implementation; the system solution should be preferred.

6.6 Export vs. import

Exporting is a more primitive operation than importing. In fact, an import could be modeled as an import request, followed by an export from the provider to the requestor. Initially we refused to have an export primitive, because we were concerned with the protection issues involved, and we didn't know how let the target context make use of the newly-available object. Recently we realized that, for some applications, export is the only natural mechanism: for instance, the Images UIMS is modeled more naturally as a window manager exporting event objects to applications, rather than applications polling the window manager for events.

The export mechanism has been implemented, but the proposed interface is not yet available.

7 Assessment of the prototype

We have already pointed out some positive and negative aspects of SOS. On the positive side: the model of elementary objects is simple and powerful;

[5] We will not discuss this point in any detail since the language interface is out of the scope of this paper.

7.4 Performance

The prototype is slow. Starting up the SOS environment takes 40 seconds of wallclock time on an otherwise unloaded, diskless Sun-3/60 with 8 Mb memory. The null application, which exits immediately, sizes 287 kbytes text, 74 kbytes data, and 87 kbytes BSS. It takes approximately 0.3 user and 0.15 system second to execute, and mallocs 445 kbytes before exiting. An application which imports a single Name Service proxy and exits, is the same size, and executes in 0.5 user plus 0.5 system seconds.

The explanation for the code size is that the kernel (37,028 bytes text + 11,820 bytes data + 8,348 bytes BSS = 57,196 bytes) and the dynamic linker (34,532 + 8,180 + 792 bytes = 43,504 bytes) are linked in with application. So are the whole standard C (198,128 bytes total) and C++ (34,376 bytes) libraries, and a few others, in case a dynamically-imported proxy needs them (SunOS 3.4 doesn't have shared libraries). The code for a few comonly-used proxy types is also linked statically to speed their importation, e.g. name service (6,536 bytes) and storage service (8,812 bytes). Thus a total of 383,388 bytes is linked by default with every executable.

The huge malloc size is attributable to the kernel pre-allocating a number of tasks, each with its own 40 kbyte stack, for handling incoming invocations. Finally the general slowness has to do with opening and using sockets, and program size, which causes swapping.

8 Future directions

Despite its limitations, SOS is a positive experience. It is a useful environment for prototyping distributed applications. Although we had implementation problems, our initial concepts have been confirmed, and we have discovered new ideas in the process. Operating system-level support for arbitrary, user-defined, medium-grained, migratory objects can be done and is useful. Our elementary-object model is both simple and powerful.

The Proxy Principle was the focal point of our initial design; with hindsight, we see that the fragmented-object concept is a more general and cleaner expression of the same idea: structuring distributed applications.

composite and persistent objects, and dynamic linking and type-checking are built on top of elementary objects; fragmented objects give structure to the universe, by extending the object concept over the net. On the negative side: SOS is not completely independent of C++; there are no persistent groups; groups are not real protection domains; the semantics of migration needs to be cleaned up.

We can mention a few more points.

7.1 Permanent pointers

We provide a library of useful pre-defined types. For instance, "permanent pointers" behave like pointers, except that they remain valid across migration [21]. Their use allows to construct a elementary object composed of many interconnected data segments (we do not currently support pointers between objects). They can be used just like pointers, but must be declared differently. This imposes a slightly unnatural programming style to C++ programmers, but it's better than banning pointers altogether.

7.2 Protection

We mentioned ealier the fact that entering a proxy does not effectively enter a protection domain. Conversely, the client importing a proxy into his context is unprotected against damage the imported code might do. We have side-stepped this issue, considering that the problem is the same with any library.

7.3 Implementation

Initially, the implementation was intended as a quick-and-dirty, throw-away prototype. The kernel is monolithic and poorly designed. The whole system is too big and fragile to experiment easily.

A limitation is that SOS is prototyped on top of Unix. This has the advantage of providing a good development environment, but the drawback is that we haven't acquired experience with implementing an operating system on the bare machine.

We must stress again the importance of the givePproxy and the re-initializer upcalls, and of pre-requisites. These give user-defined semantics to a system mechanism. Thanks to them, SOS is an extremely general system and can support many different object semantics.

The SOS prototype is currently used in our project to implement new distributed object-oriented applications, such as a reliability manager, and an original name service [11]. We are also implementing a "proxy generator" to automate the mechanical aspects of programming proxies, servers, providers, and stub procedures.

Thanks to the accumulated experience, together with Chorus-systèmes (and with support from SEPT) we have designed and implemented COOL, an object support layer in the Chorus-V3 kernel [18]. COOL is simpler and more basic than SOS: it includes only object creation, destruction, and migration, which are implemented directly in the kernel, based on the virtual-memory mechanisms of Chorus. Chorus/COOL runs directly on the bare machine; its interfaces are defined independently of any particular language. By using shared libraries and memory mapping, the size of each program file remains very modest. Persistent objects and contexts are integrated in the COOL object model.

References

[1] Jean-Pierre Banâtre, Michel Banâtre, and Florimond Ployette. An overview of the Gothic distributed operating system. Rapport de recherche 504, INRIA, March 1986.

[2] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Programming Languages and Systems*, 2(1), February 1984.

[3] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of object-oriented operating system design. Technical Report R-89-1510, Department of Computer Science, University of Illinois, Urbana, Illinois (USA), April 1989.

[4] David R. Cheriton. The V-Kernel, a software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[5] Partha Dasgupta, Richard J. Leblanc, Jr., and William F. Appelbe. The clouds distributed operating systems: Functional description, implementation details and related work. In *The 8th International Conference on Distributed Computer Systems*, pages 2–9, S. José CA (USA), June 1988. (IEEE).

[6] D. Decouchant, A. Duda, A. Freyssinet, H. Nguyen Van, M. Riveill, and X. Rousset de Pina. Guide : Un système réparti à objet. In *Actes Convention Unix 89*, pages 297–316, Paris, March 1989. AFUU.

[7] Richard P. Gabriel. The Common Lisp Object System. *AI Expert*, pages 54–65, March 1989.

[8] Philippe Gautron and Marc Shapiro. Two extensions to C++: A dynamic link editor and inner data. In *Proceeding and additional papers, C++ Workshop*, Berkeley, CA (USA), November 1987. USENIX.

[9] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[10] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[11] Jean-Pierre Le Narzul and Marc Shapiro. Un service de nommage pour un système à objets répartis. In *Actes Convention Unix 89*, pages 73–82, Paris, March 1989. AFUU.

[12] Mesaac Makpangou and Marc Shapiro. The SOS object-oriented communication service. In *Proc. 9th Int. Conf. on Computer Communication*, Tel Aviv (Israel), October–November 1988.

[13] Mesaac Mounchili Makpangou. Invocations d'objets distants dans SOS. In Guy Pujolle, editor, *De Nouvelles Architectures pour les Communications*, pages 195–201, Paris (France), October 1988. Eyrolles.

[14] Mesaac Mounchili Makpangou. *Protocoles de communication et programmation par objets : l'exemple de SOS.* PhD thesis, Université Paris VI, Paris (France), February 1989.

[15] Bertrand Meyer. Reusability: The case for object-oriented design. *IEEE Software*, pages 50–63, March 1987.

[16] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4):289–299, 1986.

[17] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4):305–367, 1988.

[18] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Frédéric Herrmann, Michel Gien, Marc Guillemont, Claude Kaiser, Pierre Léonard, Sylvain Langlois, and Willi Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4):305–370, December 1988.

[19] Marc Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198–204, Cambridge, Mass. (USA), May 1986. IEEE.

[20] Marc Shapiro. The design of a distributed object-oriented operating system for office applications. In *Proc. Esprit Technical Week 1988*, Brussels (Belgium), November 1988.

[21] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and migration for C++ objects. In *ECOOP'89*, Nottingham (GB), July 1989.

[22] SOR. Programmer's manual for SOS prototype–version 4. Rapport Technique 103, INRIA, Rocquencourt (France), December 1988.

[23] SOR. SOS reference manual for prototype V4. Rapport Technique 108, INRIA, Rocquencourt, June 1989.

[24] Bjarne Stroustrup. *The C++ Programming Language.* Number ISBN 0-201-12078-X. Addison Wesley, 1985.

[25] Bjarne Stroustrup and Jonathan E. Shopiro. A set of C++ classes for co-routine style programming. In *Proceedings and Additional Papers, C++ Workshop*, Berkeley, CA (USA), November 1987. USENIX.

From: hagmann.pa@Xerox.COM
Subject: Re: WSOS workshop
In-Reply-To: <8908090727.AA11944@blueberry.inria.fr>
To: shapiro@corto.inria.fr
Cc: hagmann.pa@Xerox.COM, hisgen@src.dec.com

I don't know when I'll be driving to Asilomar, but I can offer you a ride when I go (I both live and work in Palo Alto). I recall that things don't start until noon, so most likely I'll leave 9:00-10:00 in the morning on Wednesday. I'm not sure who is coming from SRC/Stanford, so you might find a better deal. Let me know. And give me a local contact if you are going to ride with me.

-- Bob

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique